During the T2 project showcase, the leadership identified a critical challenge: "How can we effectively monetize Discountmate while maintaining our core value proposition?" After analyzing our rich product relationship data, I've developed an SEO-driven monetization strategy and approach that creates multiple revenue streams without compromising our discount-first mission.

Key Monetization Strategies

1. **Sponsored Recommendations**
    1. Charge brands for premium placement in "Frequently Bought Together" section
    2. Example: When milk is in cart, Bega pays to have their butter appear first
2. **Enhanced Affiliate Commissions**
    1. Negotiate higher rates for driving complementary product sales
    2. Use your product relationship data to justify premium commissions
3. **Sponsored Search Results**
    1. Sell premium positioning for specific high-value keywords
    2. Use your existing keyword extraction system to identify valuable terms
4. **Retailer Bidding**
    1. Allow retailers to bid for "best price" badges
    2. Enable time-limited exclusive deals with premium commission

SEO and Product Recommendation Implementation Guide for Discountmate

1. SEO Strategy Overview

Core Technical Components

1. **Keyword Extraction & Analysis System**
    - Extract keywords from user search behavior
    - Identify high-value product associations

    - Optimize content based on search patterns
2. **Product Recommendation Engine**
    - Suggest related products based on cart items
    - Implement “frequently bought together” features
    - Personalize recommendations based on user behavior
3. **Content Optimization Framework**
    - Structure product descriptions for SEO
    - Implement schema markup for enhanced search visibility
    - Create category pages optimized for target keywords

2. Technical Implementation Plan

Backend Implementation (Python)

2.1 Keyword Extraction & Analysis System

```python
import spacy
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer
from collections import Counter

# Load NLP model
nlp = spacy.load("en_core_web_sm")

def preprocess_text(text):
    """Preprocess text by removing stopwords and lemmatizing"""
    doc = nlp(text.lower())
    return " ".join([token.lemma_ for token in doc if not token.is_stop
 and token.is_alpha])

def extract_keywords(queries, max_features=20):
    """Extract keywords using TF-IDF vectorization"""
    processed_queries = [preprocess_text(query) for query in queries]

    # Apply TF-IDF
    tfidf_vectorizer = TfidfVectorizer(max_df=0.8, max_features=max_fea
tures)
    tfidf_matrix = tfidf_vectorizer.fit_transform(processed_queries)
    keywords = tfidf_vectorizer.get_feature_names_out()

    # Create dataframe with TF-IDF scores
    tfidf_df = pd.DataFrame(tfidf_matrix.toarray(), columns=keywords)

    return tfidf_df, keywords, processed_queries

def analyze_search_patterns(keywords, processed_queries):
    """Analyze keyword frequency in search queries"""
    keyword_frequency = {}
```

```python
    for query in processed_queries:
        for keyword in keywords:
            if keyword in query:
                keyword_frequency[keyword] = keyword_frequency.get(keyword, 0) + 1

    return keyword_frequency
```

2.2 Product Recommendation Engine

```python
def build_product_associations(transaction_data):
    """Build product associations from transaction data

    Args:
        transaction_data: DataFrame with columns ['transaction_id', 'product_id']

    Returns:
        Dictionary mapping product_id to list of associated products with scores
    """
    # Group transactions by transaction_id
    transactions = transaction_data.groupby('transaction_id')['product_id'].apply(list).tolist()

    # Build co-occurrence matrix
    product_associations = {}
    for transaction in transactions:
        for product in transaction:
            if product not in product_associations:
                product_associations[product] = Counter()

            # Count co-occurrences with other products in same transaction
            for associated_product in transaction:
                if associated_product != product:
                    product_associations[product][associated_product] += 1

    return product_associations

def get_product_recommendations(product_id, product_associations, top_n=5):
    """Get product recommendations based on co-occurrence frequency"""
    if product_id not in product_associations:
        return []

    # Get most common co-occurring products
    recommendations = product_associations[product_id].most_common(top_
```

```
n)
    return [rec_id for rec_id, _ in recommendations]

def get_cart_recommendations(cart_items, product_associations, product_
metadata, top_n=5):
    """Get recommendations based on items in cart"""
    # Collect all possible recommendations from cart items
    all_recommendations = Counter()

    for item in cart_items:
        if item in product_associations:
            for rec_product, count in product_associations[item].items
():
                if rec_product not in cart_items:  # Don't recommend it
ems already in cart
                    all_recommendations[rec_product] += count

    # Get top recommendations
    top_recommendations = all_recommendations.most_common(top_n)

    # Add product metadata
    detailed_recommendations = []
    for product_id, score in top_recommendations:
        product_info = product_metadata.get(product_id, {})
        product_info['recommendation_score'] = score
        detailed_recommendations.append(product_info)

    return detailed_recommendations
```

2.3 Content Optimization Framework

```
def generate_seo_metadata(product_data, keyword_data):
    """Generate SEO metadata for product pages"""
    seo_metadata = {}

    for product_id, product in product_data.items():
        product_name = product.get('name', '')
        product_category = product.get('category', '')
        product_description = product.get('description', '')

        # Extract relevant keywords for this product
        relevant_keywords = [k for k, v in keyword_data.items()
                             if k in product_name.lower() or k in produ
ct_description.lower()]

        # Generate title (max 60 chars)
        title = f"{product_name} - {product_category} | Discountmate"
        if len(title) > 60:
            title = f"{product_name} | Discountmate"
```

```python
        # Generate description (max 160 chars)
        description = product_description[:157] + "..." if len(product_
description) > 160 else product_description

        # Generate structured data
        structured_data = {
            "@context": "https://schema.org/",
            "@type": "Product",
            "name": product_name,
            "description": product_description,
            "category": product_category,
            "offers": {
                "@type": "Offer",
                "price": product.get('price', 0),
                "priceCurrency": "AUD",
                "availability": "https://schema.org/InStock"
            }
        }

        seo_metadata[product_id] = {
            "title": title,
            "description": description,
            "keywords": relevant_keywords,
            "structured_data": structured_data
        }

    return seo_metadata
```

3. Database Schema for SEO and Recommendations

```sql
-- Products table
CREATE TABLE products (
    product_id VARCHAR(50) PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
    category VARCHAR(100),
    price DECIMAL(10, 2),
    discount_percentage DECIMAL(5, 2),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TI
MESTAMP
);

-- Keywords table
CREATE TABLE keywords (
    keyword_id INT AUTO_INCREMENT PRIMARY KEY,
    keyword VARCHAR(50) NOT NULL,
    search_frequency INT DEFAULT 0,
    importance_score DECIMAL(5, 4),
    last_updated TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_
TIMESTAMP,
```

```sql
    UNIQUE (keyword)
);

-- Product-keyword relationship
CREATE TABLE product_keywords (
    product_id VARCHAR(50),
    keyword_id INT,
    relevance_score DECIMAL(5, 4),
    PRIMARY KEY (product_id, keyword_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id),
    FOREIGN KEY (keyword_id) REFERENCES keywords(keyword_id)
);

-- Product associations (for recommendations)
CREATE TABLE product_associations (
    product_id VARCHAR(50),
    associated_product_id VARCHAR(50),
    association_score DECIMAL(10, 4),
    association_type VARCHAR(20),  -- e.g., 'frequently_bought', 'compl
ementary'
    PRIMARY KEY (product_id, associated_product_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id),
    FOREIGN KEY (associated_product_id) REFERENCES products(product_id)
);

-- User search history
CREATE TABLE user_searches (
    search_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id VARCHAR(50),
    search_query TEXT,
    search_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    result_count INT,
    processed_keywords TEXT
);
```

4. Implementation Steps

4.1 Data Collection and Processing

1. **Collect Product Data**
   - Import product catalog from Woolworths/Coles
   - Store product attributes (name, description, price, category)
   - Update pricing and discount information regularly
2. **Analyze Search Patterns**
   - Capture and store user search queries
   - Apply NLP processing to extract keywords
   - Build a keyword relevance database
3. **Analyze Transaction Data**

- Record which products are purchased together
- Build association rules for product recommendations
- Update recommendation scores based on new transactions

4.2 Frontend Implementation

1. **Product Page Optimization**

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-sc
ale=1.0">
    <title>{{product.seo_metadata.title}}</title>
    <meta name="description" content="{{product.seo_metadata.desc
ription}}">
    <meta name="keywords" content="{{product.seo_metadata.keyword
s|join(',')}}">

    <!-- Schema.org markup -->
    <script type="application/ld+json">
        {{product.seo_metadata.structured_data|json}}
    </script>
</head>
<body>
    <h1>{{product.name}}</h1>
    <div class="product-details">
        <!-- Product details here -->
    </div>

    <div class="product-recommendations">
        <h2>Frequently Bought Together</h2>
        <div class="recommendation-container">
            <!-- Render recommendation items here -->
        </div>
    </div>
</body>
</html>
```

2. **Search Results Optimization**

```html
<div class="search-results">
    <h1>Search Results for "{{search_query}}"</h1>

    <!-- Related searches -->
    <div class="related-searches">
        <p>Related searches:
            {% for related in related_searches %}
                <a href="/search?q={{related}}">{{related}}</a>{%
```

```
    if not loop.last %}, {% endif %}
                {% endfor %}
            </p>
        </div>

        <!-- Product results -->
        <div class="products-grid">
            {% for product in products %}
                <!-- Product card with structured data -->
                <div class="product-card" itemscope itemtype="https://schema.org/Product">
                    <img src="{{product.image_url}}" alt="{{product.name}}" itemprop="image">
                    <h2 itemprop="name">{{product.name}}</h2>
                    <div itemprop="offers" itemscope itemtype="https://schema.org/Offer">
                        <span itemprop="price">${{product.price}}</span>
                        {% if product.discount_percentage > 0 %}
                            <span class="discount">{{product.discount_percentage}}% OFF</span>
                        {% endif %}
                    </div>
                </div>
            {% endfor %}
        </div>
    </div>
```

4.3 API Endpoints

```python
# Flask example
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/api/recommendations/cart', methods=['POST'])
def get_cart_recommendations():
    """Get recommendations based on cart items"""
    cart_items = request.json.get('cart_items', [])

    # Get recommendations
    recommendations = get_cart_recommendations(
        cart_items,
        product_associations,
        product_metadata,
        top_n=5
    )

    return jsonify({
        'recommendations': recommendations
```

```python
    })

@app.route('/api/search', methods=['GET'])
def search_products():
    """Search products with SEO optimized results"""
    query = request.args.get('q', '')

    # Process query
    processed_query = preprocess_text(query)

    # Log search query for analysis
    if request.cookies.get('user_id'):
        log_user_search(request.cookies.get('user_id'), query, processe
d_query)

    # Get search results
    results = search_products_by_query(processed_query)

    # Get related searches
    related_searches = get_related_searches(processed_query)

    return jsonify({
        'query': query,
        'results': results,
        'related_searches': related_searches
    })
```

5. Specific Implementation for Milk → Butter/Eggs Example

5.1 Product Association Rules

To implement the specific example where adding milk to the cart triggers butter or egg recommendations:

```python
# Define manual association rules for key products
manual_associations = {
    'milk': ['butter', 'eggs', 'cereal', 'coffee'],
    'bread': ['butter', 'jam', 'peanut butter', 'cheese'],
    'pasta': ['pasta sauce', 'parmesan cheese', 'olive oil', 'ground be
ef'],
    'rice': ['chicken', 'soy sauce', 'vegetables', 'coconut milk'],
    # Add more based on common shopping patterns
}

# Combine with data-driven associations
def get_enhanced_recommendations(product_id, cart_items, manual_rules,
data_driven_rules):
    """Get recommendations combining manual rules and data-driven patte
rns"""
    recommendations = []
```

```python
    # Check manual rules first (higher priority)
    product_name = get_product_name(product_id).lower()
    for key, associated_items in manual_rules.items():
        if key in product_name:
            recommendations.extend(associated_items)

    # Add data-driven recommendations
    data_recommendations = get_cart_recommendations(
        cart_items,
        data_driven_rules,
        product_metadata,
        top_n=10
    )

    data_rec_names = [item['name'].lower() for item in data_recommendat
ions]

    # Combine recommendations (avoid duplicates)
    all_recommendations = []
    for rec in recommendations:
        if not any(rec in dr for dr in data_rec_names):
            # Find product ID for this recommendation
            rec_id = find_product_id_by_name(rec)
            if rec_id:
                rec_info = product_metadata.get(rec_id, {'name': rec.ti
tle()})
                all_recommendations.append(rec_info)

    # Add remaining data-driven recommendations
    all_recommendations.extend(data_recommendations)

    # Return top recommendations
    return all_recommendations[:5]
```

5.2 Real-time Cart Recommendations

```javascript
// Frontend JavaScript (React example)
function CartComponent() {
    const [cart, setCart] = useState([]);
    const [recommendations, setRecommendations] = useState([]);

    // When cart updates, fetch new recommendations
    useEffect(() => {
        if (cart.length > 0) {
            fetch('/api/recommendations/cart', {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
```

```
                body: JSON.stringify({
                    cart_items: cart.map(item => item.product_id)
                })
            })
            .then(response => response.json())
            .then(data => {
                setRecommendations(data.recommendations);
            });
        }
    }, [cart]);

    // Render cart and recommendations
    return (
        <div className="cart-container">
            <div className="cart-items">
                {/* Cart items */}
            </div>

            {recommendations.length > 0 && (
                <div className="cart-recommendations">
                    <h3>Frequently Bought Together</h3>
                    <div className="recommendations-grid">
                        {recommendations.map(item => (
                            <div className="recommendation-card" key={i
tem.product_id}>
                                <img src={item.image_url} alt={item.nam
e} />
                                <h4>{item.name}</h4>
                                <p>${item.price}</p>
                                <button onClick={() => addToCart(ite
m)}>Add to Cart</button>
                            </div>
                        ))}
                    </div>
                </div>
            )}
        </div>
    );
}
```

6. SEO Monitoring and Optimization

6.1 Keyword Performance Tracking

```
def track_keyword_performance(keyword_data, search_logs, conversion_dat
a):
    """Track performance of keywords in driving searches and conversion
s"""
    keyword_metrics = {}

    # Analyze search logs
```

```python
    for keyword in keyword_data:
        # Count searches containing this keyword
        search_count = sum(1 for search in search_logs if keyword in se
arch['processed_query'])

        # Count conversions from searches containing this keyword
        conversion_count = sum(1 for conv in conversion_data
                              if conv['from_search'] and keyword in con
v['search_query'])

        # Calculate conversion rate
        conversion_rate = conversion_count / search_count if search_cou
nt > 0 else 0

        keyword_metrics[keyword] = {
            'search_count': search_count,
            'conversion_count': conversion_count,
            'conversion_rate': conversion_rate
        }

    return keyword_metrics

def optimize_keywords_based_on_performance(keyword_metrics, threshold=0.
05):
    """Identify keywords to optimize based on performance"""
    # Keywords with high search volume but low conversion
    optimization_opportunities = []

    for keyword, metrics in keyword_metrics.items():
        if metrics['search_count'] > 100 and metrics['conversion_rate']
 < threshold:
            optimization_opportunities.append({
                'keyword': keyword,
                'search_count': metrics['search_count'],
                'conversion_rate': metrics['conversion_rate'],
                'potential_impact': metrics['search_count'] * (threshol
d - metrics['conversion_rate'])
            })

    # Sort by potential impact
    optimization_opportunities.sort(key=lambda x: x['potential_impact'],
 reverse=True)

    return optimization_opportunities
```

6.2 SEO Reporting Dashboard

```python
def generate_seo_dashboard_data(timeframe='last_30_days'):
    """Generate data for SEO dashboard"""
    # Collect data from various sources
```

```
    search_data = get_search_data(timeframe)
    keyword_data = get_keyword_data(timeframe)
    product_performance = get_product_performance(timeframe)

    # Top performing keywords
    top_keywords = sorted(keyword_data.items(), key=lambda x: x[1]['sea
rch_count'], reverse=True)[:10]

    # Search volume trends
    search_trends = get_search_volume_by_day(timeframe)

    # Category performance
    category_performance = get_category_performance(timeframe)

    # Recommendation conversion rate
    recommendation_performance = get_recommendation_performance(timefra
me)

    return {
        'top_keywords': top_keywords,
        'search_trends': search_trends,
        'category_performance': category_performance,
        'recommendation_performance': recommendation_performance
    }
```

7. Advanced Techniques

7.1 Semantic Product Clustering

```
from sklearn.cluster import KMeans
import numpy as np

def create_product_embeddings(products, nlp_model):
    """Create embeddings for products using NLP model"""
    embeddings = {}

    for product_id, product in products.items():
        # Combine name and description
        text = f"{product['name']} {product['description']}"

        # Get document embedding
        doc = nlp_model(text)
        embedding = doc.vector

        embeddings[product_id] = embedding

    return embeddings

def cluster_products(embeddings, n_clusters=10):
    """Cluster products based on their embeddings"""
```

```
    # Convert to numpy array
    product_ids = list(embeddings.keys())
    X = np.array([embeddings[pid] for pid in product_ids])

    # Apply KMeans clustering
    kmeans = KMeans(n_clusters=n_clusters, random_state=42)
    clusters = kmeans.fit_predict(X)

    # Map product IDs to clusters
    product_clusters = {}
    for i, product_id in enumerate(product_ids):
        product_clusters[product_id] = int(clusters[i])

    return product_clusters, kmeans.cluster_centers_

def find_related_products_by_embedding(product_id, embeddings, n=5):
    """Find related products based on embedding similarity"""
    target_embedding = embeddings[product_id]

    # Calculate similarity to all other products
    similarities = {}
    for pid, embedding in embeddings.items():
        if pid != product_id:
            # Cosine similarity
            similarity = np.dot(target_embedding, embedding) / (
                np.linalg.norm(target_embedding) * np.linalg.norm(embed
ding)
            )
            similarities[pid] = similarity

    # Sort by similarity
    related_products = sorted(similarities.items(), key=lambda x: x[1],
 reverse=True)[:n]

    return related_products
```

7.2 Seasonal and Time-Based Recommendations

```
def get_seasonal_recommendations(current_date, product_id,
base_recommendations):
    """Enhance recommendations with seasonal products"""
    # Define seasonal products
    seasonal_products = {
        # Month -> list of seasonal product IDs
        1: ["new_year_products", "summer_products"],  # January in
Australia
        2: ["back_to_school", "summer_products"],
        3: ["easter_products"],
        10: ["spring_products"],
```

```
        11: ["spring_products"],
        12: ["christmas_products", "summer_products"]
    }

    month = current_date.month

    # Get seasonal product IDs for current month
    current_seasonal_products = []
    if month in seasonal_products:
        for category in seasonal_products[month]:

current_seasonal_products.extend(get_products_by_category(category))

    # Blend recommendations
    final_recommendations = base_recommendations.copy()

    # Add seasonal products if they're related to the current product
    product_category = get_product_category(product_id)
    related_seasonal_products = [
        pid for pid in current_seasonal_products
        if is_category_related(get_product_category(pid),
product_category)
    ]

    # Replace some recommendations with seasonal ones
    if related_seasonal_products and len(final_recommendations) > 2:
        # Replace up to 2 recommendations with seasonal products
        for i in range(min(2, len(related_seasonal_products))):
            if i < len(final_recommendations) - 1:  # Keep the top
recommendation
                final_recommendations[i+1] =
related_seasonal_products[i]

    return final_recommendations.
```

Summary of Core Components

Keyword Extraction & Analysis: Uses natural language processing (spaCy) and TF-IDF vectorization to analyze user search queries, extracting valuable keywords and tracking their frequency. This system identifies what users are searching for and creates optimization opportunities.

Product Recommendation Engine: Analyzes transaction data to identify which products are frequently purchased together. It builds a "co-occurrence matrix" that maps relationships between products (like milk → butter → eggs), enabling smart recommendations.

Content Optimization Framework: Generates SEO-optimized metadata for product pages, including titles, descriptions, and structured data that helps search engines understand your content. This increases organic visibility